Introduction to Part 2

The unsupervised nature of standard SAEs often results in latent spaces where data points from different classes are intermingled, leading to poor class separability. This lack of discrimination poses challenges for subsequent classification tasks, as classifiers rely on distinct boundaries between classes to make accurate predictions. To address this issue, introducing a penalty function that encourages separation of classes in the latent space becomes essential.

By incorporating a penalty term into the training objective, we can guide the SAE to produce latent representations that are not only compact but also discriminative. This approach aligns with methodologies that aim to minimize within-class scatter and maximize between-class separation, thereby enhancing the discriminative power of the learned features. For instance, the Compact and Discriminative Stacked Autoencoder (CDSAE) framework effectively achieves this by integrating a different but similar penalty function, leading to improved classification outcomes. [1]

Another example are Variational Autoencoders (VAEs). They introduce a probabilistic framework that models the latent space as a distribution, typically Gaussian. This design facilitates the generation of new data samples by sampling from the latent distribution. [2] It's obvious here that the latent space of an auto encoder can be a useful tool in classification and generative AI.

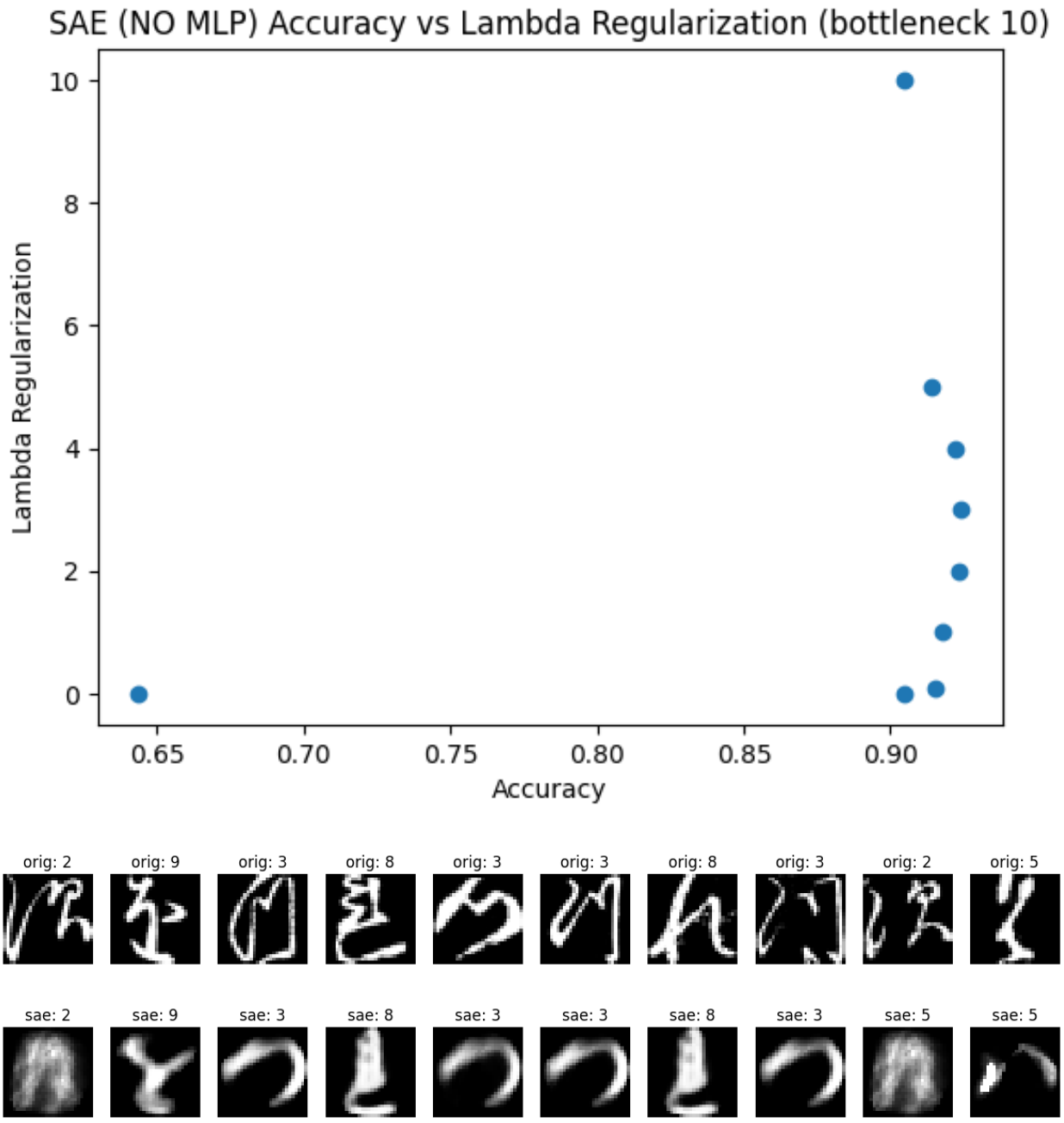
Methods

Keeping with the same methodology, the auto encoder uses the 800x200xXXX shape, but uses RAdam as it's optimizer. We note that RAdam does converge slightly faster and at higher accuracies thanks to our last project [3]

In the case of our data, we want to classify the KMNIST data set to the best of our ability. Usually, for images, CNNs offer the highest accuracy as they are able to capture features that regular MLP cannot. But, in this case, we propose using an SAE to reduce the dimensionality of the data and extract features into a smaller latent space. Instead of helping the classifier though, this can potentially cause problems as the latent space is not aware of the class designations during training.

We designate R in this project to be our penalty function, which is simply the mean distance between the target vector and the latent vector. The code defines a custom PyTorch loss function class, MSELossWithPenalty, that extends nn.Module to combine mean squared error (MSE) reconstruction loss with a penalty term aimed at enhancing class discrimination in the latent space the class sets up predefined target points for each class in the latent space using an identity matrix, where each row corresponds to a target vector for a specific class. It also takes a regularization strength parameter, lambda_reg, which controls the weight of the penalty term. In the forward method, the loss computation proceeds in two steps: first, it calculates the reconstruction loss as the MSE between the reconstructed inputs and the original inputs. Second, it computes the penalty term by measuring the Euclidean distance (using torch.linalg.norm) between the latent codes and their respective class-specific target points in the latent space. The total loss is then a weighted sum of the reconstruction loss and the penalty term, with lambda_reg scaling the penalty's contribution. This combined loss encourages accurate reconstructions while promoting latent space organization that enhances class separation.

In our experiments with this penalty term, we want to ensure that when setting the bottleneck to ten, that the SAE compares the weights appropriately. At lambda 0.001, we can see that the SAE's latent encoding itself is poor at classification, getting only a .6438 accuracy on the testing set. But, once increased to 10, the accuracy starts to drop from it's peak at .9242. The higher the lambda, the more difficult of a reconstruction as well, as seen from the poor images and poorer test loss at higher lambda values. This is expected as the penalty is making the model harder to converge.

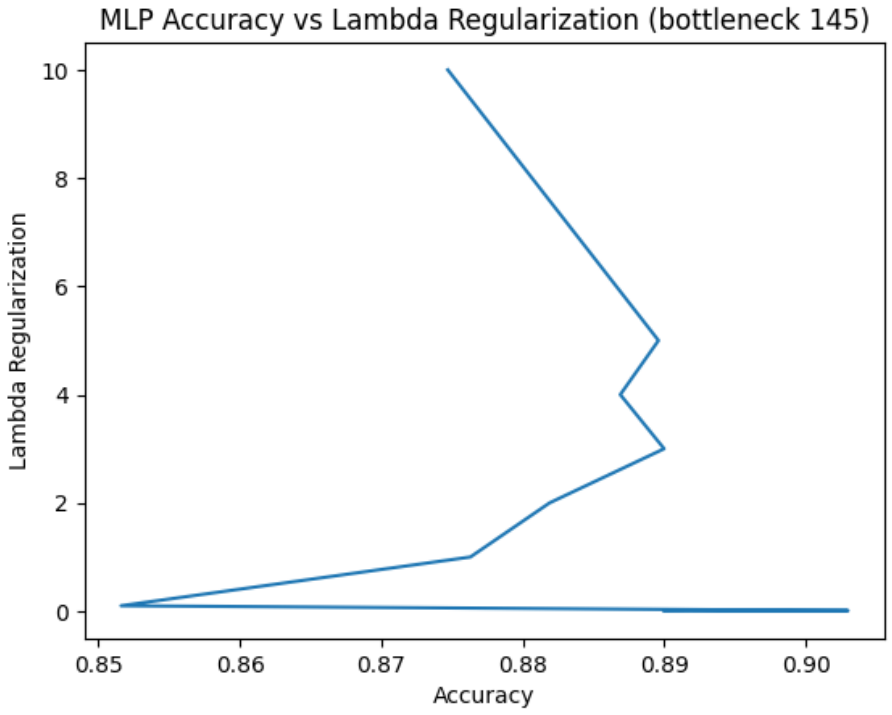


lambda 2 bottleneck 10

Results MLP 145 bottleneck

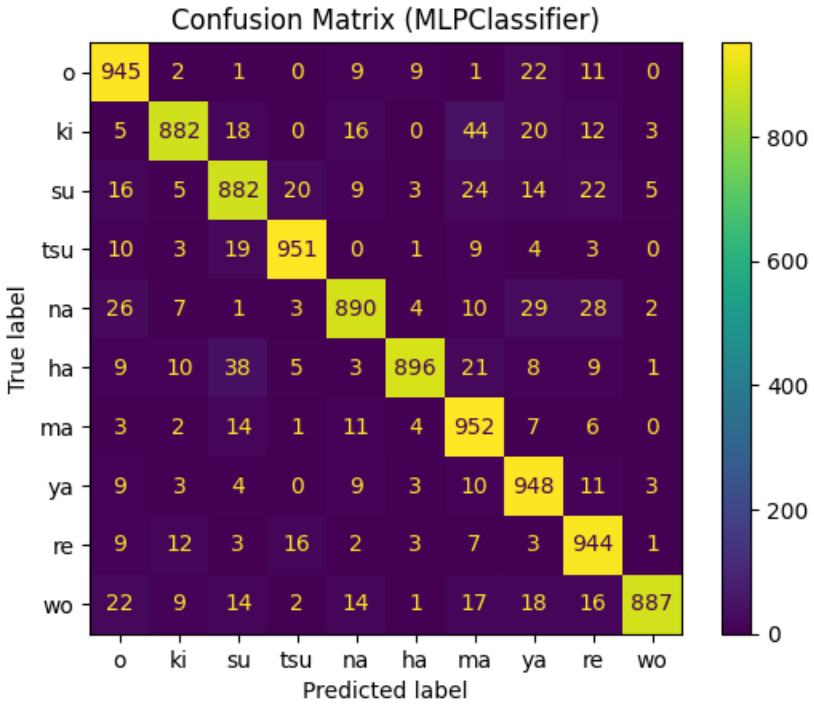
Now, doing the same training run but with the original MLP from part 1, at the bottleneck size of 145, we can see Lambda's effect on MLP's ability to classify. One thing that is making it more difficult to classify accurately is the higher bottleneck size. Because R is attempting to calculate the distance between two vectors that aren't the same size, the model is not perfectly discriminating between the classes, and is approximating. As seen when the MLP cannot reach over 90% accuracy.

Lambda 10, with bottleneck 145



Results Custom MLP with SAE

Using an MLP with an input size of 10 and an output size of 10, featuring three hidden layers with increasing and decreasing neurons (100, 200, 100), ReLU activations, a dropout layer with a 10% dropout rate, and a sigmoid activation for the final output. We were able to get a 0.9177 on the test set. The SAE had a lambda of 0.25, with a test loss (MSE) of 0.112866.



Conclusions

Keeping the MLP as it was for the 145 bottleneck size wasn't helpful in creating a better accuracy. Once the data had been encoded alongside the penalty, the penalty was much more valuable when it was in the same dimension as it's number of classes (labels). Still, the accuracy doesn't quite reach the original ~.95 from the original MLP. This is because the uncompressed data is still more accurate.

[1] https://ieeexplore.ieee.org/document/8641458
[2] https://arxiv.org/abs/1906.02691
[3] Project 1